(12) **UK Patent Application** (19) **GB** (11) **2 293 675** (13) **A**

(21) Application No 9523209.6

(22) Date of Filing **01.07.1993**

Date Lodged **10.11.1995**

(30) Priority Data
(31) **9222799** (32) **30.10.1992** (33) **GB**

(62) Derived from Application No. **9506109.9** under Section **15(4)** of the Patents Act 1977

(71) Applicant(s)
Tao Systems Limited

(Incorporated in the United Kingdom)

200 Dukes Ride, CROWTHORNE, Berkshire, United Kingdom

(51) INT CL^6
G06F 9/46

(52) UK CL (Edition O)
G4A AFN

(56) Documents Cited
GB 2128782 A **EP 0540151 A2**

(58) Field of Search
UK CL (Edition N) **G4A AFN**
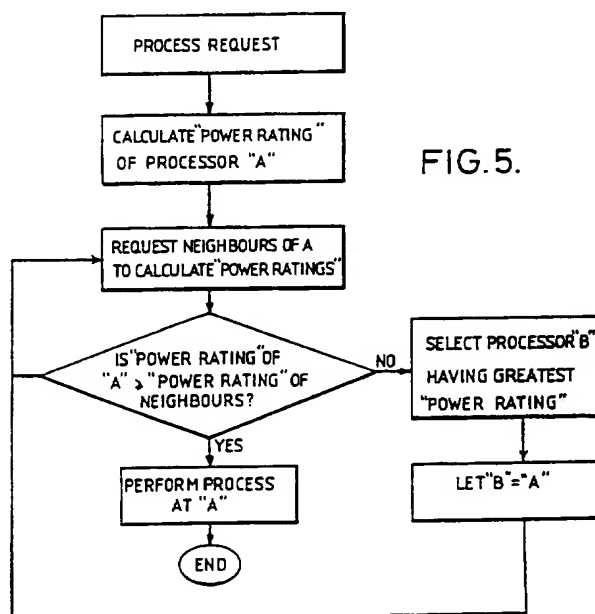INT CL^6 **G06F 9/46**

(72) Inventor(s)
Christopher Andrew Hinsley

(74) Agent and/or Address for Service
Hugh R Lambert
24 Copperkins Lane, AMERSHAM, Bucks, HP6 5QF, United Kingdom

(54) **Process allocation in a data processing system**

(57) A data processing system comprises a plurality of data processors interconnected as nodes in a network and able to perform processes in parallel, wherein the network is arranged to perform a process of process allocation in which the data processor (A) at any given node, upon receiving an instruction to perform a process, decides whether it, or a processor at one of the neighbouring nodes, is better able to perform the process, and in accordance with that decision either performs the process itself, or passes the instruction to perform the process to that neighbouring processor. The decision may be based on the local memory space or computing power available at the node.
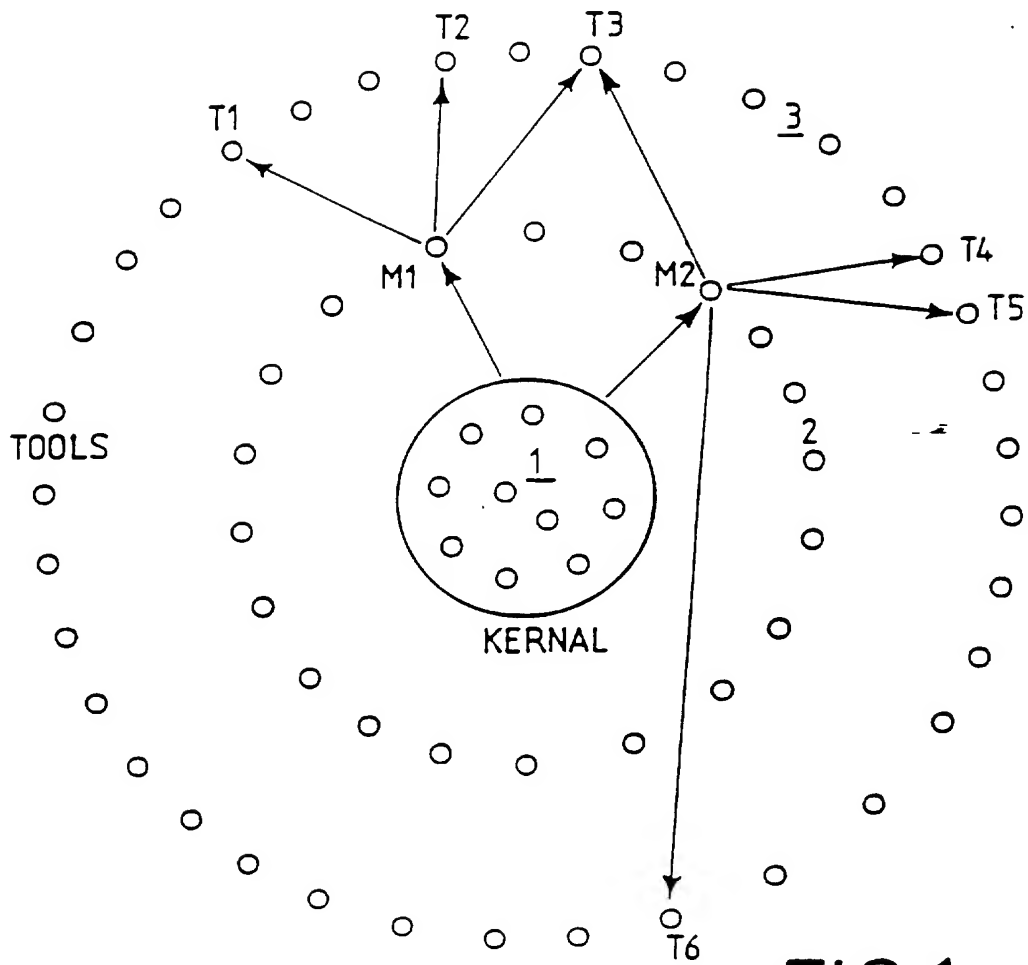
FIG.5.

GB 2 293 675 A

FIG.1

FIG.2.

$P_1 \cdots Pn$
$SP_1 \cdots SPm$

4

DBAT

$P_1 \cdots Pn$
$SP_1 \cdots SPm$

DBAT

VP

4

FIG.3.

DBAT

$P_1 \cdots Pn$
$SP_1 \cdots SPm$

4

$P_1 \cdots\cdots Pn$
$SP_1 \cdots\cdots SPm$
$L_1 \cdots\cdots L_4$

DBAT

$P_1 \cdots Pn$
$SP_1 \cdots SPm$
$L_5 \cdots L_8$

DBAT

VP

FIG.4.

DBAT

$P_1 \cdots Pn$
$SP_1 \cdots SPm$

FIG.5.

PROCESS REQUEST

CALCULATE "POWER RATING" OF PROCESSOR "A"

REQUEST NEIGHBOURS OF A TO CALCULATE "POWER RATINGS"

IS "POWER RATING" OF "A" ≥ "POWER RATING" OF NEIGHBOURS?

NO

SELECT PROCESSOR "B" HAVING GREATEST "POWER RATING"
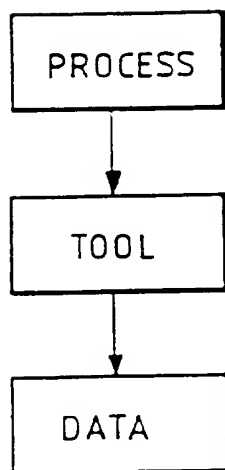
YES

PERFORM PROCESS AT "A"

LET "B" = "A"

END

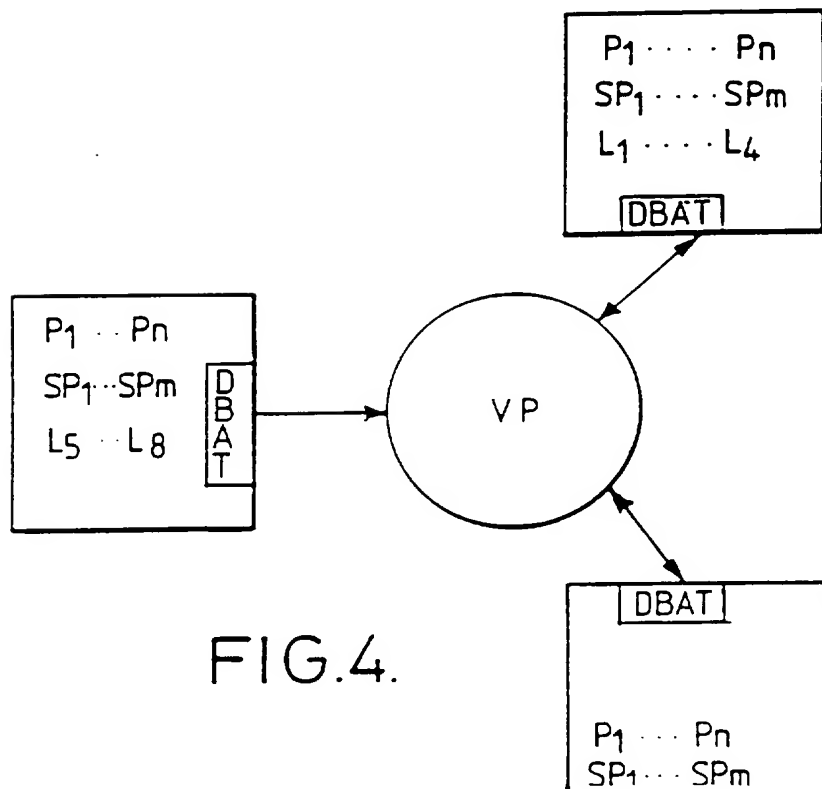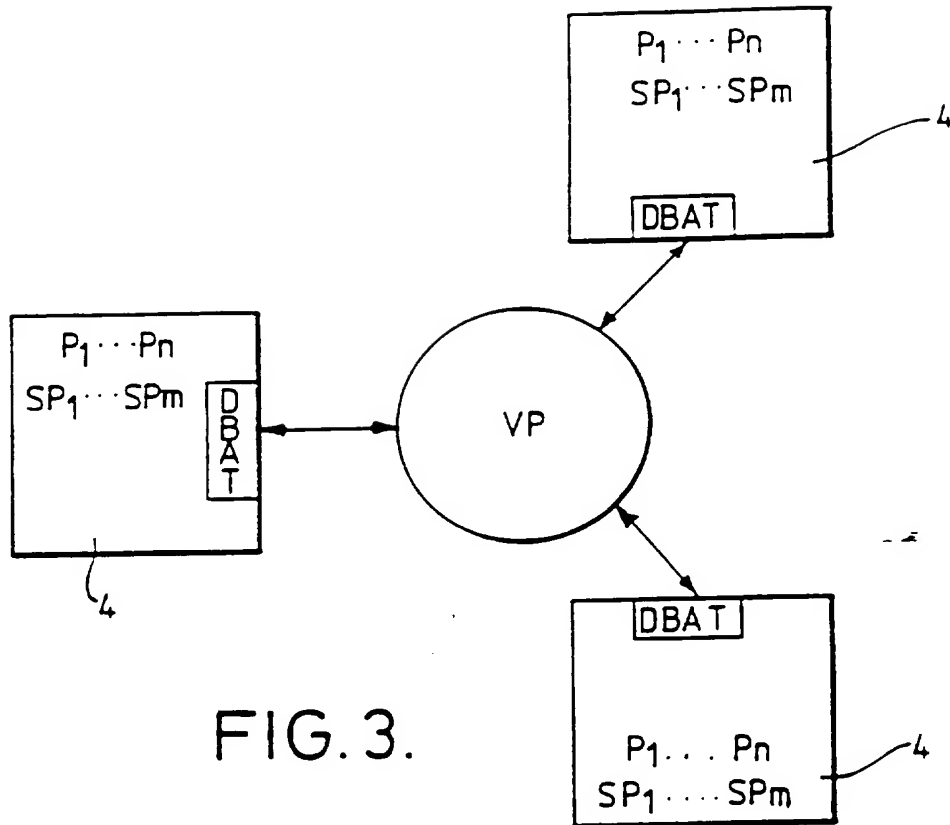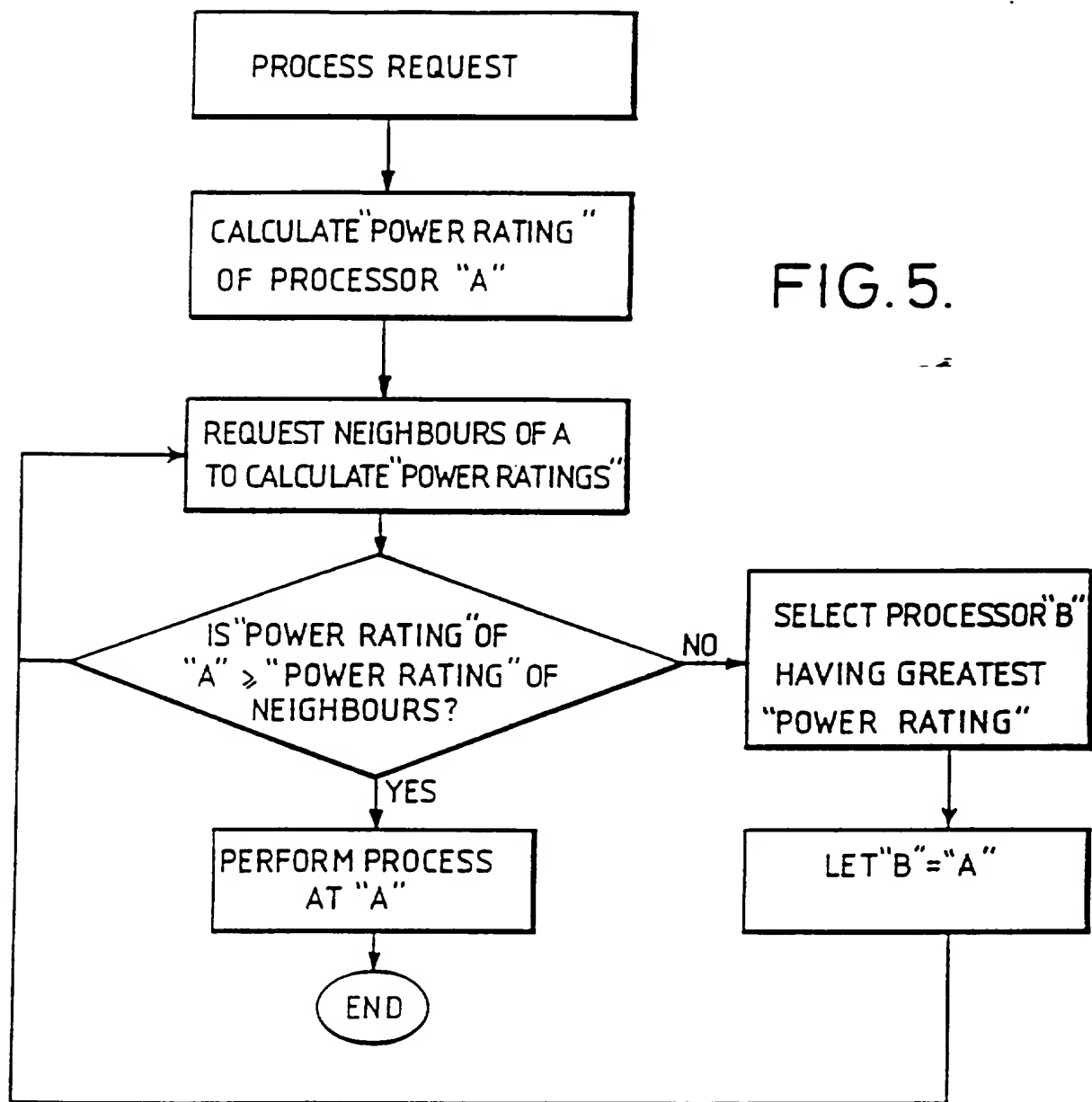## DATA PROCESSING SYSTEM AND OPERATING SYSTEM

This invention relates to data processing systems and to operating systems therefor. In particular, the invention relates to operating
5   systems for computers and computer networks.

In order to function, a computer or computer system must comprise hardware, such as one or more processors, input/output means, memory and various peripheral devices, and software in the form of both operating programs to cause the hardware to perform in
10   a certain manner and higher level programs which instruct the operating programs to perform operations. Many operating systems have been designed in attempts to obtain the best possible performance from the various processors available. Many of these operating systems are unique to one particular processor. Furthermore, they may
15   only support one or a small number of processing languages.

The present invention arose in an attempt to design an improved computer and computer operating system involving a plurality of processors capable of operating in parallel and comprising an improved means of allocating processes to be performed to individual
20   processors within the system in the most efficaceous manner.

In accordance with this invention, a data processing system is provided comprising a plurality of processors interconnected as nodes in a network and wherein the network is arranged to perform a process of process allocation in which the processor at any given node in the
25   network, upon receiving instructions to perform a process, decides whether it, or a data processor at one of the neighbouring nodes, is better able, at that time, to perform the process, and in accordance with that decision either performs the process itself or passes the instruction

to perform the process to that neighbouring processor.

In one embodiment the data processor at the first node may send a message to the data processor at each adjacent node indicative of the processing space required to execute the process, the processor at the first node also determining itself whether it has sufficient remaining space, the processor at each adjacent node being arranged to reply to the processor at the first node with information indicative of whether the space is available, and wherein the processor at the first node compares the replies with its own determination and either passes the process on to whichever of the neighbouring processors has the most space available, or takes the process itself if it has the most space available.

Alternatively, the processor with the highest "power rating" is selected, the power rating being the effective operations per second rating of the processor at any given node divided by the number of processes running at that processor and multiplied by a function of the off-chip communication speed available to the processor in question. Other methods and schemes of making the decision may be used.

Other aspects and features of the invention will be described with reference to the accompanying drawings, in which:

Figure 1 shows schematically the program organisation of a computer system;

Figure 2 shows the hierarchy of the organisation of Figure 1;

Figure 3 shows the inter-relationship between a number of associated microprocessors during setting up of the computer system;

Figure 4 shows the same relationship at a later stage during processing;

Figure 5 shows a schematic flow chart for process allocation.

The following description is of a computing network having a plurality of processors, which may be arranged either in the same vicinity, perhaps on the same wafer or chip, or spread out in a network linked by hard wiring, optical fibres, longer distance telecommunication lines, or by any other means. The system described is sufficiently flexible that any of these configurations and others besides, are appropriate. The system may alternatively include just one processor. The processor or processors are arranged to perform operations, programs and processes under the control of an operating system. The operating system enables programs to be written for an imaginary processor, known as a virtual processor, which has its own predefined language. This virtual processor enables any type of target processor to be used, or combinations of different types of processor, since the program code written for the virtual processor is only translated at load time into the native code of each processor. Thus, executable files can be transferred to any supported processor without modification or recompilation. Typical processors which are suitable for the system include the Inmos T800 Transputer, Motorola 680X0, Intel 80386/80486, TM534C40 Archimedes ARM and SUN SPARC. Other processors are of course suitable. The operating system is adapted for parallel processing such that more than one operation may be conducted simultaneously, as opposed to conventional computing systems which must pipeline data and operation so

that only one operation can be performed at any one
time. Multi-tasking and multi-user capabilities are
also included.

        The system is data flow-driven and
5   essentially involves a plurality of code segments or
tools, which are bound into a complete executable
task only at the time of loading. Thus, each of these
tools can be very short and, in themselves, almost
trivial in their simplicity. For comparison, it
10   should be noted that traditional computing binds
routines such as libraries, functions, and so on just
after the compile stage in a process known as
linking. The resulting file is a large, bulky,
complete self-contained executable image requiring
15   very little modification at load time. The file is
not portable across processors and cannot easily
adapt to changing components such as systems with
small memory capacities. Furthermore, the component
parts thereof cannot be reused for other tasks or
20   indeed changed at all. The small components used in
the present invention, which are only brought
together to form an operation at the last minute (a
just in time process) are of course completely
portable and may also be shared between two or more
25   processes at the same time, even though the
processes are performing different jobs. Some
tools are designed for a particular application
and others are designed without any particular
application in mind. With the late binding process,
30   program components can also be bound during
execution. Thus, large applications having a
(nominally) large capacity can in fact run
in a small memory capacity system and the application
can choose to have available in memory only that
35   particular code segment or segments required at any

one time, as is described below.

Data in the system is passed in the
form of messages between code components. A code
component which is capable of receiving and sending
messages is known as a process and consists of a
collection of tools and the binding logic for these
and the data being processed. The system is
illustrated schematically in Figure 1 in which there
is shown a central kernal of program code. The
kernal is one 'block' and is the basic operating
system which is used by all the processors in the
network. Typically, it is of 8 to 16 kilobytes.
Schematically illustrated outside the kernal are a
plurality of tool objects 2, any collection which can
form a process. A third level of program code is
given by the tools 3. A typical process is shown by
the arrows in the figure. Thus, a process which we
shall call P1 comprises the kernal code 1 plus two
tool objects M1 and M2. Each of these tool objects,
which may be of only a few hundred bytes long, can
utilise one or more of the tools 3. In this case,
tool object M1 utilises tools T1, T2 and T3 and tool
object M2 utilises T3 (again), T4, T5 and T6. It is
thus seen from the figure that a relatively complex
process or program can be performed by using a
plurality of the relatively small segments, which are
brought together only when necessary. More than one
process can be performed simultaneously by placing
each process upon a different processor. The actual
placement of processes occurs by an automatic method
outlined below, which method is transparent to the
user, and indeed, to the system as a whole. Thus,
optimum load balancing and communications efficiency
is ensured. As a result of this, a number of
processes that communicate can be automatically run

in parallel.

The three fundamental component objects of
the system are shown in Figure 2. These comprise
data objects, tool objects (tools) and process
objects. A process object acts as a harness to call
various tool objects. A process essentially
directs the logic flow of an activity and directs
tool objects to act on data. Thus, a process object
is at the top of the hierarchical structure shown in
Figure 2. A tool object corresponds to a traditional
functional call or sub-routine. It can only be used
by reference and must be activated by something else
such as a process object. This is similar to the
manner in which a function in C language is called by
a main body of code, the important distinction being
that in the present system each code segment, ie
tool, is completely independent. Finally, a
data object corresponds to a data file. A data file
may contain information about itself and about how it
can be processed. This information can comprise, for
example, information that points to the tool used
to manipulate the data structure and local data
relating to this file. A process can have a publicly
defined input and output, such as an ASCII data
stream. The process can thus be used by third party
processes, ie not only by the author. A process that
is not publicly available will have a private input
and output structure and thus is only free for use by
the author of a particular program, unless details of
the input/output interface are made publicly
available. One or more processes may combine to form
an application which is a complete job of work. It
should be noted that an application can run on two or
more processors.

As described above, applications can be

multi-tasked and/or parallel processed. Processes
can activate other, child, processes which can
themselves activate child (grandchild) processes, and
so on. These processes are distributed through the
available network of processors dynamically, and
without the application or the programmer knowing the
actual distribution. Communication between processes
is based on a parent-child hierarchy. There are no
"master" nodes or processors as such and any
arbitrary topology of any number of processors may be
used. Figure 5 shows one possible algorithm for
distributing processes between different processors
in the network.

      When a parent process wishes to co-process
with a child process the message is firstly passed to
a first processor, and in particular to the kernel
process on that processor. This process handles
messages to and from the processor and acts as a
process control/message switcher. The kernel
of the first processor, designated "A" then
calculates the "power rating" of the processor. The
power rating is essentially a measure of the capacity
or space within that processor to perform that
particular process. The power rating may be defined
as being the effective operations per second rating
of the processor divided by the number of processes
running on that processor multiplied by a function of
the off chip communications speed available to it.
Other methods of determining a power rating may of
course be used. For instance, a message to perform a
particular process will be accompanied by the number
of bytes required by the process. Thus the power
rating may comprise determining whether that number
of bytes are available within the processor's local
memory. This memory may be an on-chip memory for a

processor such as a transputer, or may be off-chip.
The local kernel then instructs the kernel of every
neighbouring processor in the network (remembering
that by neighbouring is meant topologically
5   neighbouring and that physically the processors may
be a large distance apart) to calculate their own
power ratings. Each of the neighbouring processors
then sends a message back to the parent processor A
indicative of the respective power rating. The mail
10   guardian of A then compares all of the power ratings
and decides if its own power rating is greater than
or equal to that of its neighbours. If so then the
mail guardian of A decides to accept the process
itself. It does this by instructing its own dynamic
15   binder and translator (see below) to instal the
child, then sends a message to the instructing
processor informing it of the unique mail box address
on the receiving processor. If the power rating of
processor A is not greater than or equal to that of
20   its neighbours then whichever processor has the
greatest power rating is chosen to accept the
process. If all power ratings are equal then
processor A will accept the process. Alternatively,
a neighbouring may be selected arbitratily. If the
25   process is allocated to another processor then
processor A sends a message to that processor saying
"take X Bytes, process name, parent mail box no.".

Having found a processor to take the child
process, the process allocation dynamics could stop
30   and the process could be performed. However, it is
more usual to repeat the same allocation sequence
from the new processor, ie, in the flow chart shown
in Figure 5, the new processor chosen, processor B,
becomes processor A and the cycle starts again from
35   the step of requesting the neighbours to calculate

their power ratings. Thus, the search for a local
processor having the minimum current activity and
thus greatest capacity to perform a process
automatically tends to flood out from a centre point
which is the originating master parent. This
substantially guarantees load balancing between the
network of processors and local communications
between tightly bound processes. Furthermore, it is
seen from the above that no master node is required
and in any network any one processor will usually
have no more than one process more than any neighbour
to perform at any time, if identical types.
Alternatively, in some embodiments the user can, if
desired, select which processor or type of processor
can perform a process. For instance, if the network
contains two types of processor, one of these types
may be better adapted to perform a particular
process, dependent upon memory capacity, for example.
The user can then specify that this process is to be
performed by a particular type of processor. It is
also seen that the process, and therefore the
programmer, has no requirement to know exactly where
it is in the network. A message can carry either
data or code between processors, allowing messages to
be in the form of runnable code.

In an alternative embodiment, the
processors may continually pass information relating
to their "power rating" between each other. This may
be passed as embedded information in messages passed
during the normal course of communication or, during
periods where no other messages are being passed between
processors, a specific exchange of power rating
information only may occur, thus using unused
communication time most efficiently. Each processor
can then be provided with a look-up table, for

example, which gives the status of its immediate
neighbouring processors and its own immediate status.
The power rating information may, for example, be
passed in the form of one or a few bytes at the end
or begining of each ordinary message passed between
processors. In this embodiment, when receiving a
request to perform a process, a processor can
immediately determine whether or not to take the
process itself or to pass it to a neighbouring
processor, and will know which one. Having
determined to pass on a request to a neighbouring
processor, the request is sent and the receiving
processor then starts the decision process again
until such a time as a processor determines that it
is best adapted to perform the process itself.

A process can be in one of five distinct
states, these are

1.   actively processing
2.   waiting for input
3.   withdrawing
4.   inactive
5.   asleep.

States 1 and 2 require no comment. State 3 occurs
when a process has received a command to withdraw.
In this state a process performs any activity which
is necessary to become inactive. This can include,
for example, processing urgent mail, sending withdraw
messages to all of its child processes, if any and
generally preparing for a convenient state in which
it can go inactive. If the process forms the top
level of an application then it will often save its
exact current state as a message in a disk file.
State 4 represents a process that is effectively

terminated. In traditional systems, this would mean
that the process in memory would cease to exist and
its memory space would be cleared for subsequent use.
In the system according to the present invention,
however, the process is marked as inactive but is not
removed unless and until its memory space is actually
required. Thus, if another process in the same local
processor references an inactive proess in state 4 it
can immediately go to state 2. State 5 is that state
of a process when it is in permanent store such as on
disk or other storage medium. When a process is
inactive in state 4 and its memory space is required
for another process, then that process goes from
state 4 to state 5, ie is stored on disk and removed
from the local processor's memory. When a process
which is in state 5 is required for a particular
local processor, that process is loaded from
permanent store. Mail may then be available for the
processor which mail is take from three sources
either from the process' state on disk, or from the
process' own header, ie information forming part of
the process, or mail from the parent process which
cause the process to be loaded in the first place.
This   can cause the process to transit   from
state 4 to state 2 and then to state 1.
        The term "disk" is introduced above.
This is a device containing a collection of data
objects "files" stored on a physical storage medium.
        As described, all process are automatically
given mailbox facilities and can read and send mail
to any other process having a known mailing address.
This process could be, for example, a child process,
the parent of the process, any named resource, for
example a disk drive or a display, any inactive
process whose full storage address is known or any

other process having *a* known mailbox address. If an
active process sends mail to an inactive process the
mail may cause the inactive process to be executed as
a child process or the mail may be stored until the
target process is awake.

A message passed between two active
processes may take the following structure:

#bits	Description
32	Message length in bytes
32	Message Type Mask — type of message, eg code, debugging information, error data
32	Message data offset — points to start of message and therefore allows for the fact that the DTM (see below) is not a fixed length
32	Next Destination Pointer (NDP) — signifies the address to which a reply should be directed
64	Originator Mailbox
64	Destination Target Mailbox (DTM) — list of process ID's
64	2nd DTM
..	(more DTM's)
..	
..	
(xxx)	message data

The message length is a byte count of the
entire length of a message.

If the Next Destination Pointer points to a
Destination Target Mailbox of "0" then no reply is
required or expected. The array of onward

destination mailboxes for messages does not imply
that a particular message is to be forwarded. The
existence of a valid DTM in the array signifies that
a reply to a message should be forwarded to that
process. In a simple case of a message requiring no
reply the DTM array will contain three values, the
originator process mailbox, the target process
mailbox, then O. Attempting to send a message to the
target ID O causes the message to be routed to the
system mail manager. A message requiring a reply to
the sender will have a DTM array consisting of four
values: originator mailbox, destination mailbox,
originators mailbox, O.

A pipe line of processes handling a stream
of data will receive a DTM array which has (number of
processing pipe) + two elements, including the
originators mailbox, plus the final destination as
one of the processes in the pipe.

'Forked' pipe schemes are possible but
require that a process actively creates the fork,
with another process possible actively joining the
fork later. Simple linear pipes need only reusable
tool objects that operate in a similar fashion to
Unix filters (as known to those skilled in the art)
but can run in parallel and can be persistent in the
network.

Forked and jointed messaging is
automatically handled when programming using the
shell with simple input-output filter tools.

Once in the memory space of a destination
process the message structure has two further
elements added before the message length:

#bits	Description
32	Forward link pointer

```
32          Backward link pointer
32          Message length in bytes
32          Message Type Mask
etc
```

5

The forward and backward link point to other messages, if any, that make up the linked list of messages for a process's incoming mailbox.

10 Mail is read by a processor in two stages, first the size of the message, then the rest of the message into a buffer allocated dynamically for the message.   This mechanism allows for a message to be redirected around a failed node or a kernel which is, for some reason, incapable of receiving a message.

15 Messages may be allowed to be distributed to more than one destination, by means of a mail group distributor. Any process which knows that it belongs to a group of processes can send a message to a mail group distributor declaring its group

20 membership. Alternatively, a process can send a message to the mail group distributor which defines all its member processes. Once informed, the mail group distributor informs all member processes of the mailbox ID which is to be used for that group. Mail

25 which is to be distributed to all members of the group is sent to the group mailbox and copies are sent to all members except the originator. A single process may belong to more than one mail group, and sub-groups may be provided. Private messages can also

30 be sent between processes in a group without all other processes being aware.

The following types of tool may be used; permanent tools form the core of the operating system. They are activated by the boot-up sequence

35 and cannot then be disabled. Every processor always

has a copy of each permanent tool available.

Semi-permanent tools are activated for
every processor by the boot process. The user can
choose which semi-permanent tools can be activated.
Once activated they cannot be deactivated.

Library tools are used as required from a
named library of tools. Once activated, they
remain cached in the memory of any processor that has
run an application that references them, until memory
constraints require their deactivation.

Application tools are either virtual or
non-virtual. Virtual ones are not necessarily active
when an application is running, but are activated as
required when a process attempts to reference them.
When not running, the virtual tool remains cached
unless the memory space is otherwise required. Thus,
automatic 'overlay' is available for tools in large
applications that cannot fit into available memory.
Non-virtual tools are loaded with an application
and in active place before it executes. Thus, they
are always available, and in memory during execution.

Each processor, or group of processors, in
the network includes a permanent tool known as the
dynamic binder and translator "DBAT". DBAT is
written in the native code of the processor, and thus
a specific version of DBAT is required for each
different supported processor. DBAT is capable of
converting the virtual processing code into the
native code of the ————————————— — processor.
Each DBAT uses a list of tools. The list
is a permanent and external tool list (PETL) which
contains information about all permanent and semi-
permanent tools (this list is identical for every
node) and all external tools such as libraries
which are currently referenced by the node, or

previously referenced and still active, or inactive yet still available, for each individual node. On receiving a command to execute a message, the object is read into the local memory area of the process. By the term processor is also meant more than one processor which is connected in a network with one DBAT. A pass is made through the object. In that pass, DBAT adds tools to the PETL.

External tools are added to the PETL in the following way; if the tool is already in the PETL, it is accepted as available, or, if the tool does not exist in the PETL it is read in, and linked into the list of available tools. If this newly installed tool contains other tool references, DBAT deals with these recursively in the same way. DBAT continues until all the external tools are available.

DBAT 'translates' the VP code of the process and newly installed tools to the target processor, inserting and correcting pointer information to tools inside the code. The unresolved virtual tool references are converted to a TRAP to the virtual tool handler inside DBAT, with a parameter pointing to the full pathname of the tool. On completion, DBAT allows the process to execute.

As each external tool is referenced by an incoming process, it has its usage flag incremented so that the tool has a value for the number of processes currently using it. As each process activates, the usage flag is decremented. When no process is using a tool, the tool has a usage flag value of zero, and can be de-activated by removal from the PETL and its memory released. This will only happen if memory space garbage collection --

occurs, allowing the system to cache tools. Semi
permanent and permanent tools are installed by DBAT
at boot up with an initial usage value of 1, thus
ensuring that they cannot be de-activated.

5   Virtual tools are called by a process while running.
A virtual tool is referenced with a pointer to the
tool's path and name, plus the tool's normal
parameters.

The virtual tool handler in DBAT checks
10   the PETL to see if the tool is already available.
If so the tool is used as for a normal external
tool. If it is absent DBAT activates the tool
(the same process as for any external tool
activated during a process start up) and then passes
15   on the tool reference. This is entirely
transparent with the exception that memory
constraints may cause a fault if even garbage
collection fails to find enough memory for the
incoming tool. Virtual memory techiques can be used
20   to guarantee this memory.

Figures 3 and 4 show, schematically, a
system in operation. Three processors (or processor
arrays) 4 are shown, schematically 'linked' to a
central 'virtual' processor VP. It is assumed that
25   each processor has a total memory, either on or off
chip. At boot up, each processor takes in DBAT and
all the other permanent $(P_1 \ldots P_n)$ and semi
permanent tools $(SP_1 \ldots SP_m)$. This state is
shown in Figure 3. All processors contain the same
30   group of permanent and semi permanent tools.

Figure 4 shows a later state. One
processor has been instructed to perform a process
which uses library tools L_1, L_2, L_3 and L_4.
It therefore has read these in. In turn, this
35   processor may read in various virtual or other

tools as required to execute a particular process. Another processor has read in tools L_5, L_6, L_7 and L_8. The final processor is not, presently, performing any process. Two or more processors may, of course, use the same library function by reading a copy of it into local memory. Once a process has ended, the library tools remain cached in the respective local memories until needed again, or until garbage collection. Of course, several processes may be run simultaneously on any processor network as described above.

As well as handling the translation of VP code to native processor code, DBAT may, in some embodiments, recognise native processor coding. Using native coding prevents portability of code but is acceptable as a strategy for hardware specific tools (commonly drivers).

It should be noted that a process or tool may be coded either in VP code or native code, and mixed source objects may be used together. However VP and native code cannot usually be mixed within a single object. A utility based in DBAT will accept VP code input and output native code for later running. This allows time critical application to be fully portable in VP 'source', but 'compiled' completely for a target system.

In order to help prevent unconstrained copying of applications, applications may retain the use of nature coded objects ability keeping the original VP coding in house for the creation of different products based on specific processors. However, use of this feature prevents applications from running on parallel systems with mixed processor types.

If an externally referenced tool is not

available then an error may be displayed, under the
action of DBAT with the known details of the missing
tool. The user can then decide whether to
temporarily replace the missing tool with another
5   one or to permanently substitute another. Tool
loss may occur by a missing or broken access
route/path or, for example, where a removable medium
such as a floppy disk is currently not accessible.
Objects can be moved on and off a network, but this
10   may often involve the removal of a whole application,
all its dependent processes etc.

The virtual processor described above is an
imaginary processor used to construct portable
"executable" code. As described above, the output of
15   VP compatible assemblers and compilers does not
require linking due to the extremely late binding
system which uses DBAT.

An 'executable' in VP code is processed by
DBAT to its host target processor at load time before
20   control is passed to the code. The VP has a 16 by 32
bit register set with an assumed 32 bit linear
address space, but not limited to 16. Registers R0
to R15 are completely general purpose. Registers such
as an IP, segment pointers, flags, etc are assumed,
25   that is, they are implemented as necessary by the
target processor. VP code is designed to be
extremely simple to 'translate' to a modern target
processor, regardless of operating philosophy. Other
word sized processors than 32 bit ones can be handled
30   easily. VP register set usage is not removed in the
final target executable code. The register set is
often implemented in memory (called ghost registers)
with the target register set being used as required to
implement the VP operation set on the memory
35   registers. This pseudo interpretation of VP

coding by a native processor may have a slight speed
disadvantage for some processors, but many benefits
overall. Processors with cached or fast memory (such
as the transputer) can actually show a performance
5   increase, particularly where the native
register set is very small, as in the transputer.
Task switching times are extremely fast using this
scheme since the state of the native processor can be
ignored, with the exception of a memory write to save
10   the position of an outgoing task's register set, and
memory read to restore the register position of the
incoming task. The transputer architecture makes
even this unnecessary. Some embodiments may have a
simple version of DBAT that does not convert VP
15   coding to native coding. These use VP macros that
are complied by a target processor macro assembler
into target executables. However certain constructs,
such as references to tools, objects, etc, are
retained for processing and fixup by DBAT. Industry
20   standard target macro assemblers are perfectly
suitable for this conversion.

The efficiency of code ultimately run on a
target processor is dependent on the two stages of
compilation and assembly by the VP compiler and DBAT
25   respectively. The ability of a compiler to reduce
the VP output to it's most efficient form is not
hampered, but obviously little or no peephole
optimisation can take place. DBAT performs peephole
optimisations on the final code produced primarily by
30   removing redundant native register/ghost register
movements that can be produced by some sequences of
VP codes.

On transputers, VP ghost registers can
result in equal or better performance than
35   traditional register use. Ghost registers are placed

in on-chip RAM on transputers, effectively producing a 'data cache' for all processes running. Up to 48 processes can use on-chip memory ghost registers on an Inmos T800 transputer. Further processes require space in off-chip memory.

The system described may include a graphical user interface for operating using windows, mice and the like. Windows can be displaced, with respect to a display means, and operated on, or their contents operated on, by means of a mouse, keyboard, or other input means.

In relation to the systems described above with reference to the accompanying drawings, the feature or features which constitute the present invention are defined and set out in the following claims.Other aspects of those systems are protected in their own right by Application No. GB 9506109.9 (GB-A-2 286 269), the parent of the present application, and by a second divsional application, Application No.   (GB-A-   ).

Finally and without prejudice to other Trade Marks used herein and not so acknowledged, the following are specifically acknowledged as Registered Trade Marks:

INMOS

TRANSPUTER

MOTOROLA

INTEL

CLAIMS

1.   A data processing system comprising a plurality of data processors interconnected as nodes in a network and able to perform processes in parallel, wherein the network is arranged to perform a process of process allocation in which the data processor at any given node, upon receiving instrusions to perform a process, decides whether it, or a processor at one of the neighbouring nodes, is better able to perform the process, and in accordance with that decision either performs the process itself, or passes the instruction to perform the process to that neighbouring processor.

2.   A system as claimed in Claim 1, wherein the first-mentioned processor determines a rating of itself and ratings of its neighbouring processors for performing the process, compares the thus deteremined ratings to determine which of the processors has the highest rating, and then either allocates the process to whichever of the neighbouring processors has the highest rating, or performs the process itself if it itself has the highest rating.

3.   A system as claimed in Claim 2, wherein, if the first-mentioned processor and one or more of the neighbouring processors have equal highest ratings, the first-mentioned processor performs the process itself.

4.   A system as claimed in Claim 2 or 3, wherein the rating of each processor is determined from the number of processes currently running on that processor.

5.   A system as claimed in Claim 4, wherein the rating of each processor is determined from the effective-operations-per-second rating of that processor divided by the number of processes running at that processor multiplied by a function of the off-chip communication speed available to that processor.

6.   A system as claimed in Claim 2 or 3, wherein the rating of each processor is determined from the available amount of local memory of that processor.

7.   A system as claimed in any of Claims 1 to 6, wherein the network is further arranged

to repeat the process of process allocation, at least once, from the processor receiving the process in the first place, with the processor to which the task was then allocated then becoming the "first" processor.

5 8.   A system as claimed in any of Claims 1 to 7, wherein, to allocate the process to the processor at one of the neighbouring nodes, the first processor instructs the neighbouring processor with data indicative of the number of bytes required, the name of the process, and the address at which the process is currently stored.

10 9.   A system as claimed in any of Claims 1 to 8, wherein information relating to the ability of the processors at the nodes to perform processes is added to other messages passed between those processors.

10.   A system as claimed in any of Claims 1 to 8, wherein information relating to the
15 ability of the processors at the nodes to perform processes is passed between pairs of processors at adjacent nodes when no other information is being passed between those processors.

20

The Patent Office

24

Application No: GB 9523209.6	**Examiner:** B.G.Western
Claims searched: 1-10	**Date of search:** 19 December 1995

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

    UK Cl (Ed.N): G4A AFN

     Int Cl (Ed.6): G06F 9/46

Other:

Documents considered to be relevant:

Category	Identity of document and relevant passage			Relevant to claims
X	GB-2128782-A	GEC	N.b. page 3	1
X,P	EP-0540151-A2	IBM	See whole document	1,7,10

DERWENT-ACC-NO: 1996-162266
DERWENT-WEEK: 199636
COPYRIGHT 2008 DERWENT INFORMATION LTD

TITLE: Operating systems for computer and computer networks interconnects processors as nodes in network for performing processes in parallel by processor deciding whether it or neighbouring node processor is better to perform process, based on local memory space or computing power

INVENTOR: HINSLEY C A

PATENT-ASSIGNEE: TAO GROUP LTD[TAOTN] , TAO SYSTEMS LTD[TAOSN]

PRIORITY-DATA: 1992GB-022799 (October 30, 1992)

PATENT-FAMILY:

PUB-NO	PUB-DATE	LANGUAGE
GB 2293675 A	April 3, 1996	EN
GB 2293675 B	August 14, 1996	EN

APPLICATION-DATA:

PUB-NO	APPL-DESCRIPTOR	APPL-NO	APPL-DATE
GB 2293675A	N/A	1995GB-023209	November 10, 1995
GB 2293675B	N/A	1995GB-023209	November 10, 1995

INT-CL-CURRENT:

TYPE	IPC DATE
CIPS	G06F9/45 20060101
CIPS	G06F9/50 20060101

RELATED-ACC-NO: 1994-128481 1994-167700 1996-162265

ABSTRACTED-PUB-NO: GB 2293675 A

BASIC-ABSTRACT:

A data processing system comprises a number of data processors interconnected as nodes in a network and able to perform processes in parallel. The network is arranged to perform a process of process allocation. The data processor at a given node decides whether it or a processor at one of the neighbouring node is better able to perform the process, upon receiving instructions to perform a process.

In accordance with the decision, the processor either performs the process itself or passes the instruction to perform the process to the neighbouring processor. The processor determines a rating of itself and rating of its neighbouring processors for performing the process. The processor compares the determined rating to determine which of the processors has the highest rating, and then the processor either allocates the process to whichever of the neighbouring processors has the highest rating or performs the process itself if it itself has the highest rating.

ADVANTAGE - Enhances operation of network. Efficient allocation process.

CHOSEN-DRAWING: Dwg.5/5

TITLE-TERMS: OPERATE SYSTEM COMPUTER NETWORK INTERCONNECT PROCESSOR NODE PERFORMANCE PROCESS PARALLEL DECIDE NEIGHBOURING BASED LOCAL MEMORY SPACE COMPUTATION POWER

DERWENT-CLASS: T01

EPI-CODES: T01-F02; T01-M02C;

SECONDARY-ACC-NO:
Non-CPI Secondary Accession Numbers: 1996-135934